

SOFTWARE TESTING AND AUTOMATION

1

COURSE CODE: 14CS604
CREDITS: 03
SEMESTER: VI
TOTAL CONTACT HOURS: 39
SEE MARKS: 50
CIE MARKS: 50
DURATION OF SEE: 03 HRS

COURSE CONTENTS

UNIT-I:

SOFTWARE PRACTICE I:

4 HRS

STYLE: NAMES, EXPRESSIONS AND STATEMENTS, CONSISTENCY AND IDIOMS, FUNCTION MACROS, COMMENTS.

SOFTWARE PRACTICE II :

4 HRS

ALGORITHMS AND DATA STRUCTURES: SEARCHING, SORTING, LISTS, TREES, HASH TABLES.

~~INTERFACES: COMMA-SEPARATED VALUES, INTERFACE PRINCIPLES, RESOURCE-MANAGEMENT, USER INTERFACES.---~~

UNIT-II:

SOFTWARE PRACTICE III:

4 HRS

PERFORMANCE: A BOTTLENECK, TIMING AND PROFILING, STRATEGIES FOR SPEED , TUNING THE CODE, SPACE EFFICIENCY, ESTIMATION.

SOFTWARE PRACTICE IV:

4 HRS

DEBUGGING: DEBUGGERS, GOOD CLUES, EASY BUGS, NO CLUES, HARD BUGS, LAST RESORTS, NON REPRODUCIBLE BUGS, DEBUGGING TOOLS, OTHER PEOPLES BUGS.

UNIT-III

SOFTWARE TESTING:

7 HRS

THE SIX ESSENTIALS OF SOFTWARE TESTING: THE STATE OF THE ART AND STATE OF THE PRACTICE, THE CLEAN SHEET APPROACH TO GETTING STARTED, ESTABLISHING A PRACTICAL PERSPECTIVE, CRITICAL CHOICES: WHAT, WHEN AND HOW TO TEST, CRITICAL DISCIPLINES, FRAMEWORKS FOR TESTING.

UNIT-IV

TESTING METHODS:

8 HRS

VERIFICATION TESTING: BASIC VERIFICATION METHODS, GETTING LEVERAGE ON VERIFICATION, VERIFYING DOCUMENTS AT DIFFERENT PHASES, GETTING THE BEST FROM VERIFICATION, THREE CRITICAL SUCCESS FACTORS FOR IMPLEMENTING VERIFICATION, RECOMMENDATION.

VALIDATION TESTING : VALIDATION OVERVIEW, VALIDATION METHODS, VALIDATION ACTIVITIES, RECOMMENDATION STRATEGIES FOR VALIDATION TESTING, CONTROLLING VALIDATION COSTS : MINIMIZING THE COST PERFORMING TESTS , MINIMIZING THE COST OF MAINTAINING THE TESTS, MINIMIZING VALIDATION TESTWARE DEVELOPMENT COSTS, RECOMMENDATIONS, TESTING TASKS, DELIVERABLES AND CHRONOLOGY, MASTER TEST PLANNING, VERIFICATION TESTING TASKS AND DELIVERABLES, VALIDATION TESTING TASKS AND DELIVERABLES. A TESTING ORPHAN – USER MANUALS, PRODUCT RELEASE CRITERIA, SUMMARY OF IEEE/ANSI TEST RELATED DOCUMENTS.

UNIT-V

MANAGING TESTING TECHNOLOGY & AUTOMATION:

8 HRS

TESTING TOOLS: CATEGORIZING TESTING TOOLS, TOOL ACQUISITION, MEASUREMENTS, USEFUL AND OTHER INTERESTING MEASURES, RECOMMENDATIONS.



ORGANIZATIONAL APPROACHES TO TESTING: ORGANIZING AND REORGANIZING TESTING, STRUCTURAL DESIGN ELEMENTS, APPROACHES TO ORGANIZING THE TEST FUNCTIONS, SELECTING THE RIGHT APPROACH: CURRENT PRACTICES, TRENDS, CHALLENGES, GUIs: WHAT IS NEW HERE, USAGE TESTING, TESTER TO DEVELOPER RATIOS, SOFTWARE MEASURES AND PRACTICES BENCHMARK STUDY.

TEXT BOOKS :

BRIAN W. KERNIGHAN AND ROB PIKE : THE PRACTICE OF PROGRAMMING, ADDISON-WESLEY, 1999.
CHAPTERS 1(EXCEPT 1.5), 2 (EXCEPT 2.3, 2.4, 2.5& 2.6), 4 (EXCEPT 4.2, 4.3 & 4.4), 5, 7.

ED KIT : SOFTWARE TESTING IN THE REAL WORLD, ADDISON – WESLEY, 1995.
CHAPTERS 1 TO 14.



UNIT 1

Compiled By
Ms. Shruthi M
Dept of CSE.

INTRO TO SOFTWARE PRACTICES - I



Why do we have to follow good software practices and what are those good practices?

Being a software programmer often we might have encountered the following situations:

- > Wasting a lot of time coding the wrong algorithm
- > Using a data structure too complicated
- > tested a program but missed on obvious problem - **Testing**
- > spent a day looking for a bug you should have found in five minutes - **Debugging**
- > needed to make a program run three times faster and use less memory - **Performance**
- > struggled to move a program from workstation to a PC and viceversa. - **Portability**
- > trying to make modest change in someone else's program –

Design alternatives, Style

- > rewritten a program but couldn't understand it.

In the world,
Enormous and intricate interfaces, constantly changing tools,
languages and systems and relentless pressure basic principles are
forgotten.

Basic programming principles (applies to all levels of computing):
Forms the bedrock of good software

Simplicity: Keep programs simple and manageable.

Clarity: makes sure they are easy to understand for people as well as machines.

Generality: working well in broad range of situations and adapt well as new situations arise.

Automation: machine do the work for us, freeing us from mundane tasks.

Good Programming: Style



Objective:

Good style is very important to good programming

- Well-written programs (fewer errors, easier to debug and modify)

Analyse This fragment of code comes from a large program written many years ago:

```
if ( (country == SING) || (country == BRNI) ||  
(country == POL) || (country == ITALY) )  
{  
    /*  
    * I f the country is Singapore, Brunei or Poland  
    * then the current time is the answer time  
    * rather than the off hook time.  
    * Reset answer time and set day of week.  
    */  
    ....
```




Interpretations that might result by analysing the code:

- Careful written, formatted, commented.
- But
 - What is the relationship between Singapore, Brunei , Poland and Italy?
 - Why isn't Italy mentioned in the comment?
 - Since comment and code differs, so one must be wrong or both are
 - Only code gets executed so may be its right
 - Comment didn't get updated when the code did
 - If you need to maintain the code, more information is required as comment does not specify relationship between these countries.

Practice of programming:

Write software that works , avoiding trouble spots and weaknesses.

Purpose of style in programming:

(In contrast to poorly written code well-written programs)

1. Make the code **easy to read for yourself and others.**
2. Than writing a program just that gets syntax right, fixes the bugs and makes run fast enough programs written, **are read not only by computers but also by programmers.**
3. Well written program is **easier to understand and to modify.**
4. Code should be **simple and clear, straightforward logic, natural expression, conventional language use, meaningful names, neat formatting, helpful comments, avoiding clever tricks and unusual constructions.**

Distinguish bad code from good code:

#define ONE 1

#define TEN 10

#define TWENTY 20

Consider : If an array of TWENTY elements must be made larger, everywhere the name also will have to be changed accordingly.

Or the code when written as

#define INPUT_MODE 1

#define INPUT_BUFSIZE 10

#define OUTPUT_BUFSIZE 20

Whereas here the #defines Indicates role of specific value in the program

Names



- Naming conventions – **easier to understand your own code and code written by others.**
- The Longer the program – more important is the **Choice of good, descriptive and systematic names.**
- Variable or function name – labels an object and conveys information about its purpose.
- Broader the scope of variable depending upon context of use, more information should be conveyed by its name.
- Name must be
 - Informative
 - Concise
 - Memorable
 - Pronounceable (if possible)

1. Use descriptive names for globals, short names for locals



- Global variables (functions, classes, structures)
 - names must be long enough and descriptive enough
 - include brief comment with declaration of each global
 - descriptive names suggest their role in the program.
- Local variables
 - shorter names (in conventional ways)

```
int npending = 0; //current length of input queue  
    can use numPending or num_pending also.
```

```
int n; //may be sufficient  
int npoints; // is fine  
int numberOfPoints; //is overkill
```

Compare:

```
for (theElementIndex = 0; theElementIndex < numberOfElements;  
theElementIndex++)  
    elementArray[theElementIndex] = theElementIndex;
```

To

```
for (i = 0; i < nelems; i++)  
    elem[i] = i ;
```

- Programmers are often encouraged to use long variable names regardless of context

Some naming conventions and local customs:

nodep -> names that begin or end with p for pointers

Initial capital letters -> for Globals

all capitals -> for CONSTANTS

2. Be consistent



- Give **related things related names that show their relationship and highlight their difference.**

Member names in Java class is wildly inconsistent

```
class UserQueue {  
    int noOfItemsInQ, frontOfTheQueue, queueCapacity;  
    public int noOfUsersInQueue() {...}  
}
```

- Word “queue” appears as Q, Queue, queue.
- Since queues can only be accessed from a variable of type UserQueue, member names need not mention “queue” at all.
queue.queueCapacity // is redundant



This version is better:

```
class UserQueue {  
    int nitems, front, capacity;  
    public int nusers() {. . .}  
}
```

- since it leads to statements like
 `queue.capacity++;`
 `n = queue.nusers();`
- No clarity is lost

3. Use active names for functions



- Function names must be based on **active verbs** , perhaps **followed by nouns**.

```
now = date.getTime();  
putchar('\n');
```

- Function that return a boolean(true or false) should be named so that the return is unambiguous.

```
if(checkoctal(c)) ... // does not indicate which value is true which is false
```

```
if( isoctal(c)) ... //makes clear that function returns true if argument is octal and false if not.
```

4. Be accurate



- **Misleading name can lead to mystifying bugs**
- Name not only labels, conveys information to the reader.

```
#define isoctal( c ) (( c ) >= 0 && ( c ) <= '8' )  
implementation is wrong
```

instead of proper

```
#define isoctal( c ) (( c ) >= 0 && ( c ) <= '7' )
```

//sensible names can sometimes disguise a broken implementation.

Expressions and Statements



- Write Expressions and Statements in a way that **makes their meaning as transparent as possible.**
- **Write the clearest code** that does the job.
- **Use spaces around operators to suggest grouping.**
- **Format to help readability.**

1.Indent to show structure

- **Consistent indentation style** helps make a **program structure self-evident.**

Badly formatted:

```
for (n++; n < 100; field[n++]='\0');  
*i = '\0'; return '\n';
```

Little better:

```
for (n++; n < 100; field[n++]='\0')  
;  
*i = '\0';  
return '\n';
```

- Even better is to put the assignment in the body and separate the increment, so the Loop takes a more conventional form and is thus easier to grasp:

```
for (n++; n < 100; n++)  
    field[n] = '\0';  
*i = '\0';  
return '\n';
```

2. Use the natural form for expressions

- **Write expressions as you might speak them aloud**
- Conditional expressions that include negations are always hard to understand:

```
if(!(block_id < actblks) || !(block_id >= unblocks))
```

...

- Each test is stated negatively, though there is no need for either to be.

Turning the relations around let us state the tests positively:

```
if((block_id >= actblks) || (block_id < unblocks))
```

...

Makes the code read more naturally.

- **Remember DeMorgan's Laws**

```
if(!( r=='n' || r=='N' ))
```

```
if( r!='n' && r!='N' )
```

3. Parenthesize to resolve ambiguity

- Specifies grouping – can be **used to make the intent clear** even when they are not required.

```
if((block_id >= actblks) || (block_id < unblocks))
```

...

Parenthesis can be omitted here as relational operators have higher precedence over logical operators.

- When mixing unrelated operators it is a good idea to parenthesize.

```
while ((c = getchar()) != EOF)
```

.....

Bitwise operators & and | have lower precedence than relational ==

```
if (x & MASK == BITS) //correct but ugly
```

..... means if (x & (MASK == BITS)) // incorrect

so to make the intent clear expression is as if ((x & MASK) == BITS) //
better



Even if parenthesis is not necessary, grouping is difficult to grasp as in this example:

```
leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

When used it is easier to understand

```
leap_year = ( ( y % 4 == 0 ) && ( y % 100 != 0 ) || ( y % 400 == 0 ) );
```

4. Break up complex expressions



Expression given below is very compact and packs too many operations in a single statement:

```
*x += (*xp = (2*k < (n-m) ? c[k+1] : d[k--]));
```

Its easier to grasp when broken into several pieces:

```
if ( 2*k < n-m )
```

```
    *xp = c[k+1];
```

```
else
```

```
    *xp = d[k--];
```

```
*x += *xp;
```


5. Be clear



- Write clear code not clever code.

Consider the code below:

```
subkey = subkey >> (bitoff – (bitoff >> 3) << 3));
```

Original expression is equivalent to

```
subkey = subkey >> ( bitoff & 0x7 );
```

Programmers who are experienced make it short n write it as:

```
subkey >>= bitoff & 0x7;
```

Often clearer code is shorter, proper criterion here is the ease of understanding.

- Consider the following:

child = (!LC && !RC) ? 0: (LC?RC:LC);

Difficult to follow.

The longer version is clearer to follow:

```
if( LC == 0 && RC == 0) //okay
    child = 0;
else if ( LC == 0 ) //even better
    child = RC;
else
    child = LC;
```

- The ?: is fine for replacement of four lines of if-else with one, but **not general replacement for conditional statements.**

6. Be careful with side effects

- Operators like ++ have side effects, besides returning a value they modify the value of underlying variable.
- **Actions of retrieving the value and updating may not happen at the same time.**

```
str[i++] = str[i++] = ' ';
```

intends to store blanks at the next two positions in str. But i may be updated only once.

So break into two statements:

```
str[i++] = ' ';
```

```
str[i++] = ' ';
```

Likewise `arr[i++] = i;` // if i is initially 3, array element ,might set it to 3 or 4.

```
arr[ i ] = i;
```

```
i++;
```



Arguments to a function are evaluated before the function is called.
yr is not read first

```
scanf("%d %d", &yr, &profit[yr]);
```

```
scanf("%d",&yr);           // better  
scanf("%d", &profit[yr]);
```

Consistency and Idioms



- Consistency **leads to better programs.**
 - Same computation if is done the same way every time it appears, any variation suggests a genuine difference.
-
1. **Use a consistent indentation and brace style**
 - Indentation shows structure, which is the best style?
 - When braces are present should it be in same line of if or next?
 - Layout of programs as is argued topic, but specific style is much less important than its consistent application.
 - **Pick one style, use it consistently.**

- Like parenthesis, **braces can resolve ambiguity and occasionally make the code clearer.**

```
if (month == FEB) {  
    if (year%4 == 0)  
        if (day > 29)  
            legal=FALSE;  
        else  
            if (day > 28)  
                legal = FALSE;  
}
```



Here the indentation is misleading, since else is actually attached to the line `if (day > 29)` and the code is wrong.

When one if immediately follows another, always use braces:

```
if (month == FEB) {  
    if (year%4 == 0) {  
        if (day > 29)  
            legal=FALSE;  
        } else {  
            if (day > 28)  
                legal = FALSE; }  
}
```

- Still easier if there is a **variable to hold number of days in february.**

```
if (month == FEB) {
```

```
    int nday;
```

```
    nday = 28;
```

```
    if (year%4 == 0)
```

```
        nday = 29;
```

```
    if (day > nday)
```

```
        legal = FALSE;
```

```
}
```

- If you work on program you didn't write, preserve the style you find there.
- When you make a change don't use your own style.
- Because programs consistency is more important.

2. Use Idioms for Consistency

- Conventional way that experienced programmers write common pieces of code.

Form of loop:

```
i = 0;  
while (i <= n-1)  
    array[i++] = 1.0;
```

```
for (i = 0; i < n; )  
    array[i++] = 1.0;
```

```
for (i = n; --i >= 0; )  
    array[i] = 1.0;
```

But Idiomatic form is:

```
for (i = 0; i < n; i++)  
    array[i] = 1.0;
```


- For walking along a list in C

```
for (p = list; p != NULL; p = p->next)
```

...

- For Infinite loop


```
for (;;) 
```

...

```
while (1)      // is also popular
```

...

- Indentation should be idiomatic too



```
for (  
    ap = arr;  
    ap < arr + 128;  
    *ap++ = 0;  
)  
{  
    ;  
}
```

Standard way:

```
for (ap = arr; ap < arr+128; ap++)  
    *ap = 0;
```



- Nesting an assignment inside a loop condition:

```
while ((c = getchar()) != EOF)  
    putchar(c);
```

```
do {  
    c = getchar();  
    putchar(c);  
} while (c != EOF);
```

- ◆ Only use do-while loop when the loop body must be executed at least once.

3. Use else-ifs for Multi-Way Decisions



```
if (argc == 3)
    if ((fin = fopen(argv[1], "r")) != NULL)
        if ((fout = fopen(argv[2], "w")) != NULL) {
            while ((c = getc(fin)) != EOF)
                putc(c, fout);
            fclose(fin); fclose(fout);
        } else
            printf("Can't open output file %s\n", argv[2]);
    else
        printf("Can't open input file %s\n", argv[1]);
    else
        printf("Usage: cp inputfile outputfile\n");
```

//Instead it is better to use an else-if and rewrite



```
if( argc != 3 )  
    printf( "Usage: cp inputfile outputfile\n" );  
  
else if( ( fin=fopen( argv[1], "r" )) == NULL )  
    printf( "Can't open input file %s\n", argv[1] );  
  
else if( ( fout=fopen( argv[2], "w" )) != NULL )  
    printf( "Can't open output file %s\n", argv[2] );  
  
else {  
    while( (c=getc( fin )) != EOF )  
        putc( c, fout );  
    fclose( fin ); fclose( fout );  
}
```

Use else-ifs for Multi-Way Decisions



// This form is somewhat better with comments, there is increase in size but increase in clarity also.

```
? switch (c) {  
?   case '-':  sign = -1;  
?   case '+':  c = getchar();  
?   case '.':  break;  
?   default:   if (!isdigit(c))  
?               return 0;  
? }
```

```
switch (c) {  
?   case '-':  
?       sign = -1;  
?       /* fall through */  
?   case '+':  
?       c = getchar();  
?       break;  
?   case '.':  
?       break;  
?   default:  
?       if (!isdigit(c))  
?           return 0;  
?       break;
```

Use else-ifs for Multi-Way Decisions

// Much better is sequence of else-if's



```
if (c == '-') {  
    sign = -1;  
    c = getchar();  
} else if (c == '+') {  
    c = getchar();  
} else if (c != '.' && !isdigit(c)) {  
    return 0;  
}
```

//Acceptable use of fall-throughs, no comments are required.

/* ok without comment */

case '0':

case '1':

case '2':

...

break;

1.4 Function Macros



- ◆ Avoid function macros
- ◆ Parenthesize the macro body and arguments

Avoid Function Macros



- ◆ Macros have been used to avoid the overhead of function calls – this is no longer necessary
 - In C++, inline functions renders macros unnecessary
 - ? #define isupper(c) ((c) >= 'A' && (c) <= 'Z')
 - Parameter c occurs twice in the body of macro.
 - ? while (isupper(c = getchar()))
 - Each time an input character is greater than or equal to A it will be discarded and another char read to be tested against Z.
- // Correct form is
while ((c = getchar()) != EOF && isupper(c))



```
?#define ROUND_TO_INT(x) ((int) ((x)+(((x)>0)?0.5:-0.5)))
```

```
.....
```

```
? size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

Multiple evaluation causes a pblm.

- Can house many instructions in a single function and can be called instead of a macro.

Parenthesize the Macro Body and Argument



◆ If you insist on using function macros, be careful.

? `#define square(x) (x) * (x)`

The expression

? `1/square(x)`

Will be evaluated as

? `1/(x) * (x)`

//This is better with parenthesis

`#define square(x) ((x) * (x))`

Comments



- ◆ Don't belabor the obvious
- ◆ Comment functions and global data
- ◆ Don't comment bad code, rewrite it
- ◆ Don't contradict the code
- ◆ Clarify don't confuse

Don't Belabor the Obvious

Comments **should not report self evident information.**

```
? /*  
?  * default  
?  */  
? default:  
?     break;
```

```
? /* return SUCCESS */  
? return SUCCESS;
```

```
? zerocount++; /* Increment zero entry counter */
```

```
? /* Initialize "total" to "number_received" */  
? node->total = node->number_received;
```

Don't Belabor the Obvious

```
? while ((c = getchar()) != EOF && isspace(c))  
?     ;                               /* skip white space */  
?   if (c == EOF)                     /* end of file */  
?     type = endoffile;  
?   else if (c == '(')                 /* left paren */  
?     type = leftparen;  
?   else if (c == ')')                 /* right paren */  
?     type = rightparen;  
?   else if (c == ';')                 /* semicolon */  
?     type = semicolon;  
?   else if (is_op(c))                 /* operator */  
?     type = operator;  
?   else if (isdigit(c))               /* number */
```

```
// Some actions are obvious here.
```

Comment Functions and Global Data



Brief summary can aid understanding

```
struct State { /* prefix + suffix list */
    char *pref[NPREF]; /* prefix words */
    Suffix *suf;        /* list of suffixes */
    State *next;        /* next in hash table */
}
```

```
// random: return an integer in the range [0..r-1].
int random(int r)
{
    return (int) (Math.floor(Math.random()*r));
}
```

Don't Comment Bad Code, Rewrite it



If comment outweighs the code, then fix the code.

```
? /* If "result" is 0 a match was found so return true (nonzero).  
? Otherwise, "result" is non-zero so return false (zero). */  
? #ifdef DEBUG  
? printf("*** isword returns !result = %d\n", !result);  
? fflush(stdout);  
? #endif  
? return(!result);
```


Don't Comment Bad Code, Rewrite it



```
#ifdef DEBUG
printf("*** isword returns matchfound = %d\n",matchfound);
fflush(stdout);
#endif
return matchfound;
```

Don't Contradict the Code



- ◆ Most comments agree with the code when they are written, but as bugs are fixed and the program evolves, the comments are often left in their original form, resulting in disagreement with the code.

```
? time(&now);  
? strcpy(date, ctime(&now));  
? /* get rid of trailing newline character copied from ctime */  
? i = 0;  
? while (date[i] >= ' ') i++;  
? date[i] = 0;
```

Don't Contradict the Code

```
? time(&now);  
? strcpy(date, ctime(&now));  
? /* get rid of trailing newline character copied from ctime */  
? for (i=0; date[i] != '\n'; i++)  
?     ;  
? date[i] = '\0';
```

```
time(&now);  
strcpy(date, ctime(&now));  
/* ctime() puts newline at end of string; delete it */  
date[strlen(date)-1] = '\0';
```

Clarify, don't Confuse



```
? int strcmp(char *s1, char *s2)
? /* string comparison routine returns -1 if s1 is */
? /* above s2 in ascending order list, 0 if equal */
? /* 1 if s1 below s2 */
? {
?     while (*s1 == *s2) {
?         if (*s1 == '\0') return(0);
?         s1++; s2++;
?     }
?     if (*s1 > *s2) return(1);
?     return(-1);
? }

/* strcmp: return < 0 if s1 < s2, > 0 if s1 > s2, 0 if equal */
/*      ANSI C, section 4.11.4.2 */
```

Summary

- ◆ “good style should be a matter of habit”

SOFTWARE PRACTICES - II



ALGORITHMS AND DATA STRUCTURES

Data Structures and algorithms

- **Algorithms**
 - Outline, the essence of a computational procedure, step by step instruction
- **Program**
 - An implementation of an algorithm in some computer programming languages
- **Data Structures**
 - **Goal:** *organize data*
 - **Criteria:** facilitate efficient
 - storage of data
 - retrieval of data
 - manipulation of data
- **Process:**
 - select and design appropriate data types

Data may be organized in many different ways. the logical and mathematical model of a particular organization of data is called **Data structure**

Introduction



- The study of algorithms and data structures is one of the foundations of computer science, a rich field of elegant techniques and sophisticated mathematical analyses.
- Good algorithm or data structure – makes it possible to solve a problem in seconds that could otherwise take years.
- Areas like graphics, databases, parsing, numerical analysis, and simulation, the ability to solve problems depends critically on state-of-the-art algorithms and data structures.
- Every program depends on algorithms and data structures, but few programs depend on the invention of brand new ones. Even within an intricate program like a compiler or a web browser, most of the data structures are arrays, lists, trees, and hash tables.



- **Choose appropriately algorithms and data structures available** and know how to choose among the various alternatives available.
- There are only a handful of basic algorithms that show up in almost every program-primarily **searching and sorting**-and even those are often included in libraries.
- Almost every data structure is derived from a few fundamental ones.

Searching



- **Arrays are best way of storing static tabular data.**
- Compile-time initialization makes it cheap and easy to construct such arrays.

In a program to detect words we can write

```
char *flab[] = {  
    "actual1y",  
    "just",  
    "quiten,  
    "really".  
    NULL  
};
```



- Search routine needs to know how many elements are in the array.
- One way to tell it is to **pass the length as an argument**;
- Another, used here, is to **place a NULL marker at the end of the array**.

```
/* lookup: sequential search for word i n array*/  
int lookup(char *word, char *array[])  
{  
    int i;  
    for (i = 0; array[i] != NULL; i++)  
        if (strcmp(word, array[i]) == 0)  
            return i;  
    return -1;  
}
```



- This search algorithm is called ***sequential search*** because it looks at each element in turn to see if it's the desired one.
- When the **amount of data is small**, sequential search is fast enough.
- There are standard library routines to do sequential search for specific data types; for example, functions like **strchr** and **strstr** search for the first instance of a given character or substring in a C or C++ string.
- The Java **String** class has an **indexOf** method.
- And the generic C++ **find** algorithms apply to most data types.
- If such a function exists for the data type you've got, use it.



- Sequential search is easy but the **amount of work is directly proportional to the amount of data to be searched**; doubling the number of elements will double the time to search if the desired item is not present.
- This is a **linear relationship**-run-time is a linear function of data size- so this method is also known as *linear search*.



- An array of **more realistic size** from a program that parses HTML, which defines textual names for well over a hundred individual characters:

typedef struct Nameval Nameval ; //alternative names to existing types, so that we can use Nameval to declare variables for the structure

```
struct Nameval {  
    char *name;  
    int value;  
};
```



// To declare an array consisting of word and value of type Nameval we use the following

```
/* HTML characters, e. g. AElig is ligature of A and E. */  
/* Values are Unicode/ISO10646 encoding. */  
Nameval htmlchars [] = {  
    "AElig" , 0x00~6,  
    "Aacute", 0x00~1,  
    "Aci rc" , 0x00~2,  
    /* ... */  
    "zeta", 0x03b6,  
};
```



- For a larger array like this, it's more efficient to use ***binary*** search.
- The binary search algorithm is an orderly version of the way we look up words in a dictionary.
- Check the middle element.
- If that value is bigger than what we are looking for, look in the first half; otherwise, look in the second half.
- Repeat until the desired item is found or determined not to be present.
- For binary search, the table must be sorted.

- A binary search function for this table might look like this:

/ lookup: binary search for name i n tab; return index */*

```
int lookup(char *name, Nameval tab[], int ntab)
{
    int low, high, mid, cmp;
    low = 0;
    high = ntab - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        cmp = strcmp(name, tab[mid].name);
        if (cmp < 0)
            high = mid - 1;
        else if (cmp > 0)
            low = mid + 1;
        else/* found match */
            return mid; }
    return -1; /* no match */ }
```



- Putting all this together. to search **html chars** we write

```
half = lookup("frac12", htmlchars, NELEMS(htmlchars)) ;
```

to find the array index of the character 1/2.

```
printf ("The HTML table has %d words\n" , NELEMS(htmlchars)) ; //  
NELEMS is a macro used to find the size of the array
```



- Binary search **eliminates half the data at each step.**
- The number of steps is therefore **proportional to the number of times we can divide n by 2 before we're left with a single element.**
- Ignoring roundoff, this is \log
- If we have **1000** items to search, linear search takes up to 1000 steps, while binary search takes about 10;
- if we have a million items, linear takes a million steps and binary takes 20.
- The more items, the greater the advantage of binary search.
- Beyond some size of input (which varies with the implementation), binary search is faster than linear search.

Sorting



- Binary search works only if the elements are sorted. If repeated searches are going to be made in some data set, it will be profitable to sort once and then use binary search.
- If the data set is known in advance, it can be sorted when the program is written and built using compile-time initialization. If not, it must be sorted when the program is run.
- One of the best all-round sorting algorithms is quicksort, which was invented in 1960 by C. A. R. Hoare.
- Quicksort is a fine example of how to avoid extra computing.
- It works by partitioning an array into little and big elements.

Quicksort

- Quicksort pros [advantage]:
 - Sorts **in place**
 - Sorts $O(n \lg n)$ in the **average case**
 - Very efficient in practice , it's quick
- Quicksort cons [disadvantage]:
 - Sorts $O(n^2)$ in the **worst case**
 - And the worst case doesn't happen often ... **sorted**



pick one element of the array (the "pivot").

partition the other elements into two groups:

"little ones" that are less than the pivot value, and

"big ones" that are greater than or equal to the pivot value.

recursively sort each group.

- When this process is finished, the array is in order.
- Quicksort is **fast** because once an element is known to be less than the pivot value, we don't have to compare it to any of the big ones; similarly. big ones are not compared to little ones.
- This makes it much **faster than the simple sorting methods** such as insertion sort and bubble sort that compare each element directly to all the others.

This **quicksort** function sorts an array of integers:



```
/* quicksort: sort v[0]..v[n-1] into increasing order a/  
void quicksort(int v[], int n) {  
    int i, last;  
    if (n <= 1) /* nothing to do */  
        return ;  
    swap(v, 0, rand() % n) ; /* move pivot elem to v[0] */  
    last = 0;  
    for (i= 1; i < n; i++) /* partition */  
        if (v[i] < v[0])  
            swap(v, ++last, i);  
    swap(v, 0, last); /* restore pivot */  
    quicksort(v, last) ; /* recursively sort */  
    quicksort(v+last+1, n-last-1) ; /* each part */  
}
```



```
/* swap: interchange v[i] and v[j] */  
void swap(int v[], int i, int j)  
{  
    int temp;  
    temp = v[i] ;  
    v[i] = v[j];  
    v[j] = temp;  
}
```

- Partitioning **selects a random element as the pivot**, swaps it temporarily to the front, then sweeps through the remaining elements, exchanging those smaller than the pivot ("little ones") towards the beginning (at location last) and big ones towards the end (at location i).

Consider array $v[] = \{35, 13, 67, 45, 33, 78, 12\}$ $n=7$

$\text{swap}(v, 0, \text{rand()} \% n)$ when done, randomly choose 33 n swap

$v[] = \{33, 13, 67, 45, 35, 78, 12\}$ //pivot element comes to the first position.

$\text{last} = 0, i=1$ if($v[1] < v[0]$) then $\text{swap}(v, 1, 1)$ i.e., the output will be

$v[] = \{33, 13, 67, 45, 35, 78, 12\}$

$\text{last} = 1, i=2$ if($v[2] < v[0]$) then swap ...but here we continue with next value of i

$\text{last} = 1, i=3$ if($v[3] < v[0]$) then swap...no swapping here

$\text{last} = 1, i=4$ if($v[4] < v[0]$) then swap... no swapping here

$\text{last} = 1, i=5$ if($v[5] < v[0]$) then swap ... no swapping here

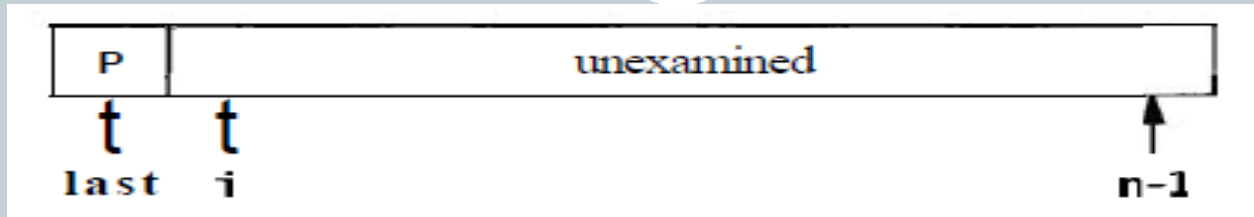
$\text{last} = 1, i=6$, if($v[6] < v[0]$) $\text{swap}(v, 2, 6)$ the output will be

$v[] = \{33, 13, 12, 45, 35, 78, 67\}$

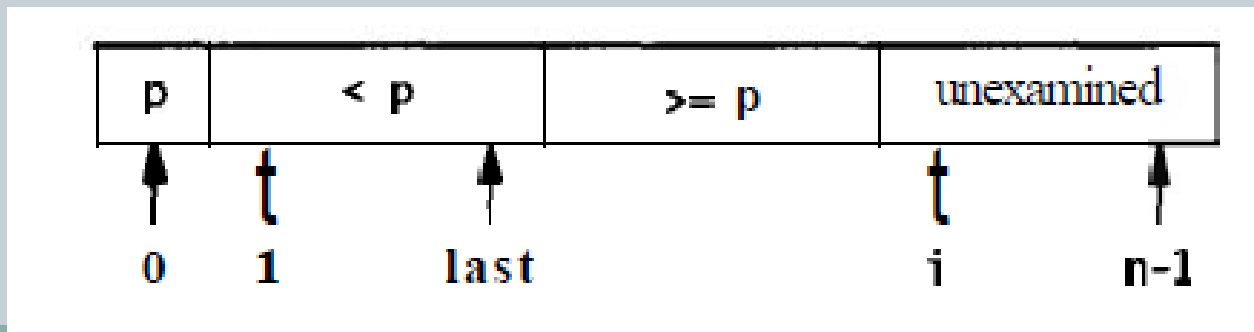
$\text{swap}(v, 0, \text{last})$ i.e., $\text{swap}(v, 0, 2)$ o/p is $v[] = \{12, 13, 33, 45, 35, 78, 67\}$

Then sort recursively

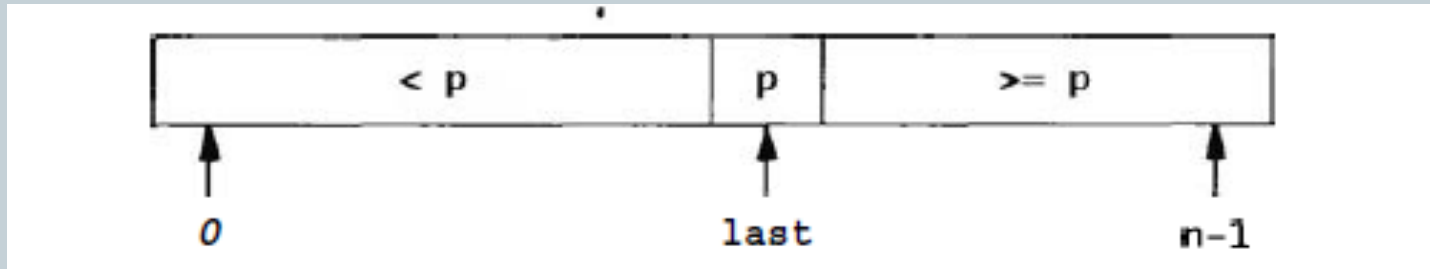
- At the beginning of the process, just after the pivot has been swapped to the front, $\text{last} = 0$ and elements $i = 1$ through $n-1$ are unexamined:



- At the top of the for loop, elements 1 through last are strictly less than the pivot, elements $\text{last}+1$ through $i-1$ are greater than or equal to the pivot, and elements i through $n-1$ have not been examined yet.
- Until $v[i] \geq v[0]$, the algorithm may swap $v[i]$ with itself; this wastes some time but not enough to worry about.



- After all elements have been partitioned, element **0** is swapped with the last element to put the pivot element in its final position; this maintains the correct ordering.
- Now the array looks like this:



- The same process is applied to the left and right sub-arrays; when this has finished, the whole array has been sorted.

How fast is quicksort?

- In the best possible case, the first pass partitions n elements into **two groups of about $n/2$ each**.
- the **second level partitions two groups, each of about $n/2$ elements, into four groups each of about $n/4$** .
- The next level partitions four groups of about $n/4$ into eight of about $n/8$ and so on.

- This goes on for about \log

- On the average, it does only a little more work. It is customary to use base 2 logarithms; thus we say that **quicksort takes time proportional to $n \log n$** .
- This implementation of quicksort is the clearest for exposition, but it has a weakness.
- If **each choice of pivot splits the element values into two nearly equal groups**, our analysis is correct.
- But if the split is uneven too often, the run-time can grow more like n^2 .

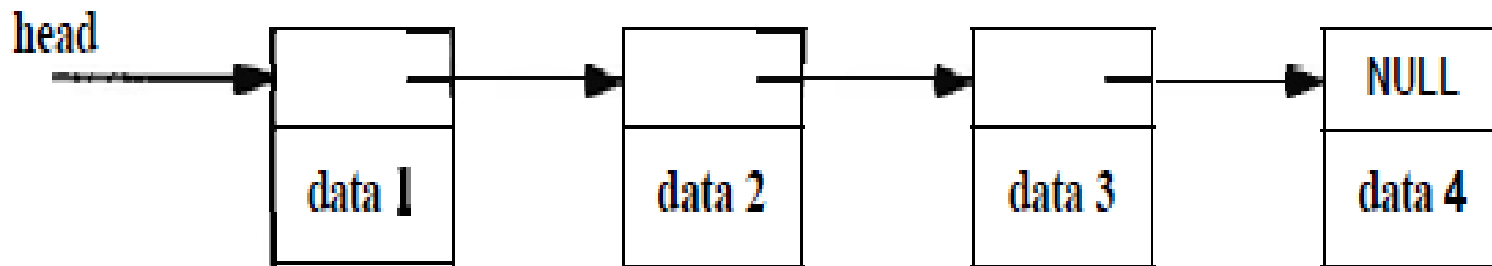


- **Our implementation uses a random element as the pivot to reduce the chance that unusual input data will cause too many uneven splits.** But if all the input values are the same, our implementation splits off only one element each time and will thus run in time proportional to n^2
- **The behaviour of some algorithms depends strongly on the input data.**
- Perverse or unlucky inputs may cause an otherwise well-behaved algorithm to run extremely slowly or use a lot of memory.
- In the case of quicksort, although a simple implementation like ours might sometimes run slowly, more sophisticated implementations can reduce the chance of pathological behaviour to almost zero.

Lists in C



- Next to arrays, lists are the most common data structure in typical programs.
- ***A singly-linked list*** is a set of items, each with data and a pointer to the next item.
- The **head** of the list is a pointer to the first item and the **end of the list** is marked by a null pointer.
- This shows a list with four elements:



There are several important differences between arrays and lists.

- First, **arrays have fixed size** but a list is always exactly the **size it needs to be to hold its contents, plus some per-item storage overhead to hold the pointers**.
- Second, **lists can be rearranged by exchanging a few pointers**. which is cheaper than the block move necessary in an array.
- Finally, when items are inserted or deleted the other items aren't moved; if we store pointers to the elements in some other data structure, they won't be invalidated by changes to the list.
- These differences suggest that if the set of items will change frequently, particularly if the number of items is unpredictable, a list is the way to store them;

By comparison, an **array is better for relatively static data**.



There are a handful of fundamental list operations:

- add a new item to the front or back,
 - find a specific item,
 - add a new item before or after a specific item, and perhaps
 - delete an item.
-
- The simplicity of lists makes it easy to add other operations as appropriate.
 - Rather than defining an explicit **List** type, the usual way lists are used in C is to start with a type for the elements, such as our HTML **Nameval**, and add a pointer that links to the next element:



typedef struct Nameval Nameval ; // Nameval can be used for creating structure variables

```
struct Nameval {  
    char *name;  
    int value ;  
    Nameval *next; /* in list */  
};
```

It's difficult to initialize a non-empty list at compile time, so, unlike arrays, lists are constructed dynamically.

First, we need a way to construct an item.

Use a suitable function to allocate memory for constructing a new item:



```
/* newitem: create new item from name and value */
```

```
    Nameval *newitem(char *name, int value)
```

```
    {
```

```
        Nameval *newp;
```

```
        newp = (Nameval *) malloc (sizeof (Nameval )) ;
```

```
        newp->name = name;
```

```
        newp->value = value ;
```

```
        newp->next = NULL;
```

```
        return newp;
```

```
    }
```

```
// malloc is a memory allocator, assuming here that it never report failure.
```

To add each new element to the front:



```
/* addfront: add newp to front of listp */  
Nameval *addfront(Nameval *listp, Nameval *newp)  
{  
    newp->next = listp ; // newp is the node created using newitem function  
    return newp;  
}
```

Function that updates a list must return a pointer to the new first element, which is stored in the variable that holds the list.

Typical use of function addfront is:

```
nvlist = addfront (nvlist, newitem("smiley",0x263A);
```

This design **works even if the existing list is empty.**

Adding an item at the end of the list, $O(n)$, as we need to walk the list to find the end:



```
/* addend: add newp to end o f listp */
Nameval *addend(Nameval *listp , Nameval *newp)
{
    Nameval *p;
    if (listp == NULL)
        return newp;
    for (p = listp ; p->next != NULL; p = p->next)
        ;
    p->next = newp;
    return listp ;
}
```

If we want to make addend an $O(1)$ operation, separate pointers must be maintained for the end of list, which is an overhead. So stick to the simple style.

To search for an item with a specific name, follow the next pointers:



```
/* lookup: sequential search for name in listp */  
Nameval *lookup(Nameval *listp, char *name)  
{  
    for (; listp != NULL; listp = listp->next)  
        if (strcmp(name, listp->name) == 0)  
            return listp ;  
    return NULL; /* no match */  
}
```

Takes $O(n)$ time, even if the list is sorted to walk along the list to get to particular element.

Binary search does not apply to lists.

To destroy a list:

/* freeall : free all elements of listp */

void freeall (Nameval *listp)

{

Nameval *next ;

for (; listp != NULL; listp = next) {

next = listp->next;

/* assumes name is freed elsewhere */

free (listp) ;

}

}

//Memory cannot be used after it has been freed, so we must save listp->next in a local variable called next, before freeing element pointed by listp.

? for (; listp != NULL; listp = listp->next) {

? free(listp);

To delete a single element from a list:

Nameval *delitem(Nameval *listp , char *name)

```
{
    Nameval *p, *prev;
    prev = NULL;
    for (p = listp ; p != NULL; p = p->next) {
        if (strcmp(name, p->name) == 0) {
            if (prev == NULL)
                listp = p->next;
            else
                prev->next = p->next;
            free (P);
            return listp ; }
        prev = p;
    }
    printf ("del item: %s not in list", name) ;
    return NULL; /* can't get here */
}
```

- List structures and operations account for vast majority of applications.



- Doubly linked list requires more overhead, but finding last element and deleting the current element are $O(1)$ operations.
- Besides being suitable for operations where there are insertions and deletions in the middle, lists are good for managing unordered data of fluctuating size, when they tend to be LIFO as in a stack.
- They make more effective use of memory than arrays do when there are multiple stacks that grow and shrink independently.
- If you must combine frequent update with random access, however, it would be wiser to use a less insistently linear data structure, such as a tree or hash table.

Trees



- A tree is a hierarchical data structure that **stores a set of items** in which **each item has a value, may point to zero or more others, and is pointed to by exactly one other.**
- The ***root*** of the tree is the sole exception; **no item points to it.**
- There are many types of trees that reflect complex structures, such as **parse trees that capture the syntax of a sentence or a program, or family trees that describe relationships among people.**

Binary Search Trees:

- A node in a binary search tree has a **value** and **two pointers**, left and right, that point to its children.



- In a binary search tree, the values at the nodes define the tree: **all children to the left of a particular node have lower values, and all children to the right have higher values.**

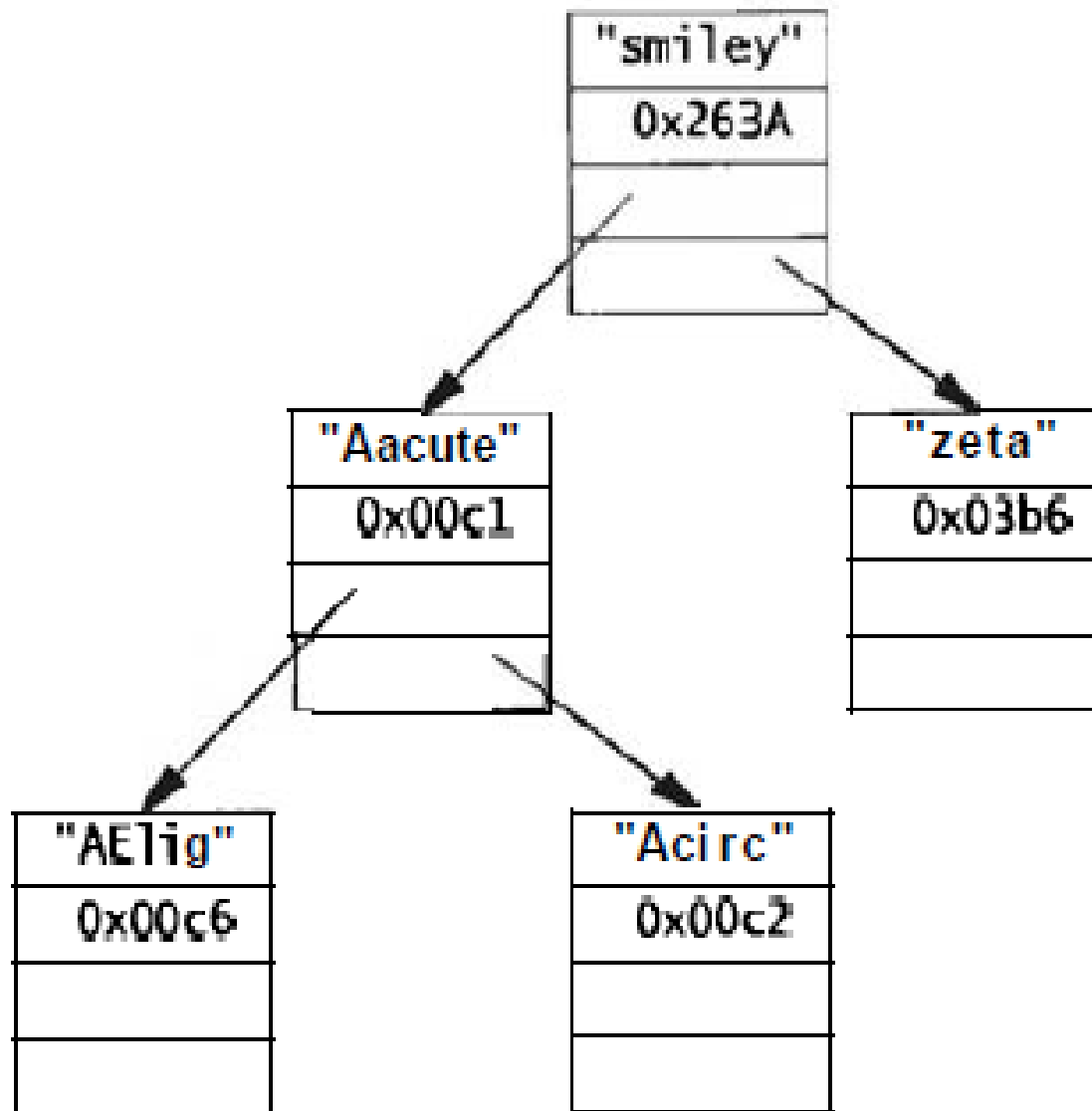
The tree version of Nameval is straightforward:

```
typedef struct Nameval Nameval;
```

```
struct Nameval {  
    char *name;  
    int value ;  
    Nameval *left; /* lesser */  
    Nameval *right; /* greater */  
};
```

The *lesser* and *greater* comments refer to the properties of the links: left children store lesser values, right children store greater values.

Binary Search Tree of Namevals





- With multiple pointers to other elements in each node of a tree, many operations that take time $O(n)$ in lists or arrays **require only $O(\log n)$ time in trees.**
- Binary search tree is constructed by **descending into the tree recursively, branching left or right as appropriate, until we find the right place to link in the new node**, which must be a properly initialized object of type **Nameval**: a name. a value. and two null pointers.
- The new node is added as a leaf, that is, it has no children yet.

```
/* insert : insert newp in treep, return treep */  
Nameval *insert(Nameval *treep, Nameval *newp)  
{
```

```
    int cmp;
```

```
    if (treep == NULL)
```

```
        return newp;
```

```
    cmp = strcmp(newp->name, treep->name);
```

```
    if (cmp == 0)
```

```
        weprintf ("insert: duplicate entry %s ignored",  
                  newp->name) ;
```

//prints error msg prefixed with the word warning does not terminate
the program unlike eprintf

```
    else if (cmp < 0)
```

```
        treep->left = insert(treep->left, newp) ;
```

```
    else
```

```
        treep->right = insert(treep->right, newp) ;
```

```
    return treep;
```

```
}
```



// In list insertion does not check for duplicate items because that requires searching the list which is $O(n)$ operation rather than $O(1)$.

- A tree in which each **path from the root to a leaf has approximately the same length is called balanced** (elements have to be in random order).
- Advantage of **searching a balanced tree** is that searching it for an item is an $O(\log n)$ process like binary search.



```
/* lookup: look up name in tree treep */
Nameval *lookup (Nameval *treep , char *name)
{
    int cmp;
    if (treep == NULL)
        return NULL;
    cmp = strcmp(name, treep->name);
    if (cmp == 0)
        return treep;
    else if (cmp < 0)
        return lookup(treep->left , name) ;
    else
        return lookup(treep->right, name) ;
}
```



Analyse the code for lookup and insert:

- Both use the idea of binary search : divide and conquer, the origin of logarithmic-time performance.
- These routines are recursive.
- If they are written as iterative algorithm they will be even more similar to binary search.



```
/* nrlookup: non-recursively look up name in tree treep */
Nameval *nrlookup(Nameval *treep, char *name)
{
    int cmp;
    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0)
            return treep;
        else if (cmp < 0)
            treep = treep->left;
        else
            treep = treep->right;
    }
    return NULL;
}
```

- When nodes are to be processed in sorted order, visiting the left subtree is first before visiting the right subtree – **Inorder traversal**.

Left subtree, root node, right subtree.

- Insert items into a tree, allocate an array of right size , use in-order traversal to store them in the array in sequence.

- **Postorder traversal** – invokes operation on current node after visiting the children

Left subtree, right subtree, root node

- Used when the operation of node depends on subtrees below it.

Eg: computing height of a tree (take max height of each subtree and add one), measuring total storage.

- B-trees (has very high branching) are generally used to maintain information on secondary storage.



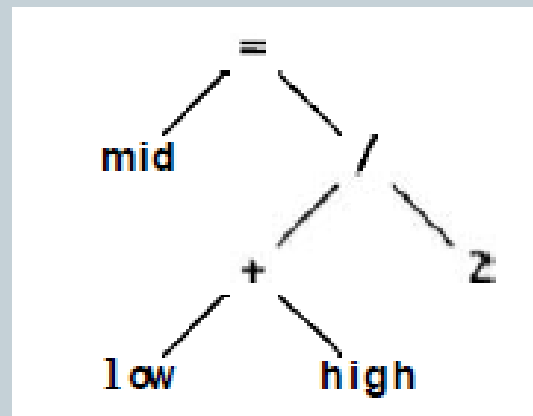
Day to day programming one common use of trees:
To represent the structure of a statement or expression.

For example, the statement

`mid = (low + high) / 2;`

can be represented by the ***parse tree*** is as below:

To evaluate the tree, do a post-order traversal and perform the appropriate operation at each node.



Hash Tables

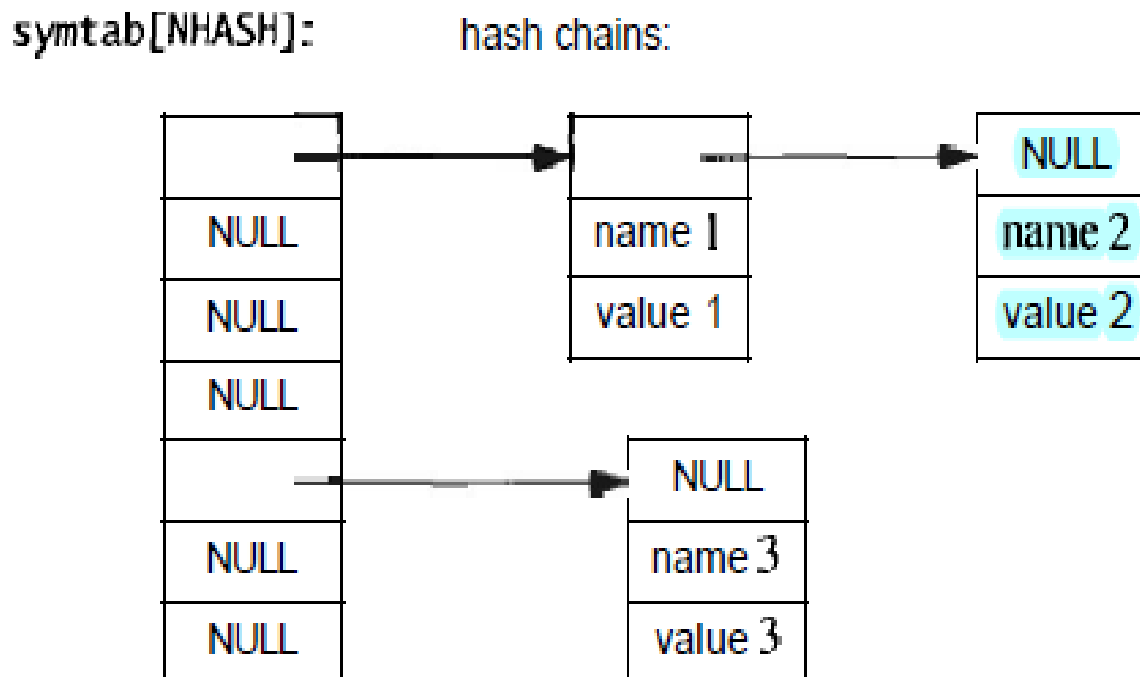


- Hash tables are one of the great inventions of computer science. They **combine arrays, lists, and some mathematics to create an efficient structure for storing and retrieving dynamic data.**
- Typical application is a symbol table. which associates some value (the data) with each member of a dynamic set of strings (the keys).
- Compiler almost certainly uses a hash table to manage information about each variable in your program.
- Web browsers may well use a hash table to keep track of recently-used pages,
- Connection to the Internet probably uses one to cache recently-used domain names and their **IP** addresses.

- The idea is to pass the key through a hash function to generate a hash value that will be evenly distributed through a modest-sized integer range.



- The hash value is used to index a table where the information is stored.
- In C and C++ the usual style is to associate with each hash value (or "bucket") a list of the items that share that hash, as this figure illustrates:



- The hash function is pre-defined and an appropriate size of array is allocated, often at compile time.
- Each element of the array is a list that chains together the items that share a hash value.

A hash table is an array of lists, the element type is the same as for a list:

```
typedef struct Nameval Nameval ;  
struct Nameval {  
    char *name;  
    int value ;  
    Nameval *next; /* in chain */  
};
```

```
Nameval *symtab [NHASH] ; /* a symbol table */
```

lookup/insert routine:

/* lookup: find name in symtab, with optional create */

Nameval* lookup(char *name, int create, int value)

{

int h;

Nameval *sym;

h = hash(name) ;

for (sym = symtab[h]; sym != NULL; sym = sym->next)

if (strcmp(name, sym->name) == 0)

return sym;

if (create) {

sym = (Nameval *) malloc (sizeof (Nameval)) ;

sym->name = name;

sym->value = value;

sym->next = symtab[h];

symtab[h] = sym;

}

return sym; }